# Object Oriented Programming (part 2)

# Agenda

- Inheritance
- Fields/Methods in Extended Classes
- Constructors in extended classes
- Inherited object construction
- Overloading and Overriding Methods
- Polymorphism
- Type compatibility

# Agenda

- Type conversion
- `protected` members
- Object: the ultimate superclass

# Inheritance

- *Inheritance*: you can create new classes that are built on existing classes. Through the way of inheritance, you can reuse the existing class's methods and fields, and you can also add new methods and fields to adapt the new classes to new situations
- Subclass and superclass have a IsA relationship: an object of a subclass IsA(n) object of its superclass

# Inheritance

- "is a" relationship
  - Inheritance
- "has a" relationship
  - Composition, having other objects as members
- Example

**`Employee`** "is a" **`BirthDate;`** **`//Wrong!`**

**`Employee`** "has a" **`Birthdate;`** **`//Composition`**

# Definitions

- A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).
- The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

# Definitions

- Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance).

# Definitions

- Every class is an extended (inherited) class, whether or not it's declared to be. If a class does not declared to explicitly extend any other class, then it implicitly extends the `Object` class

# Inheritance

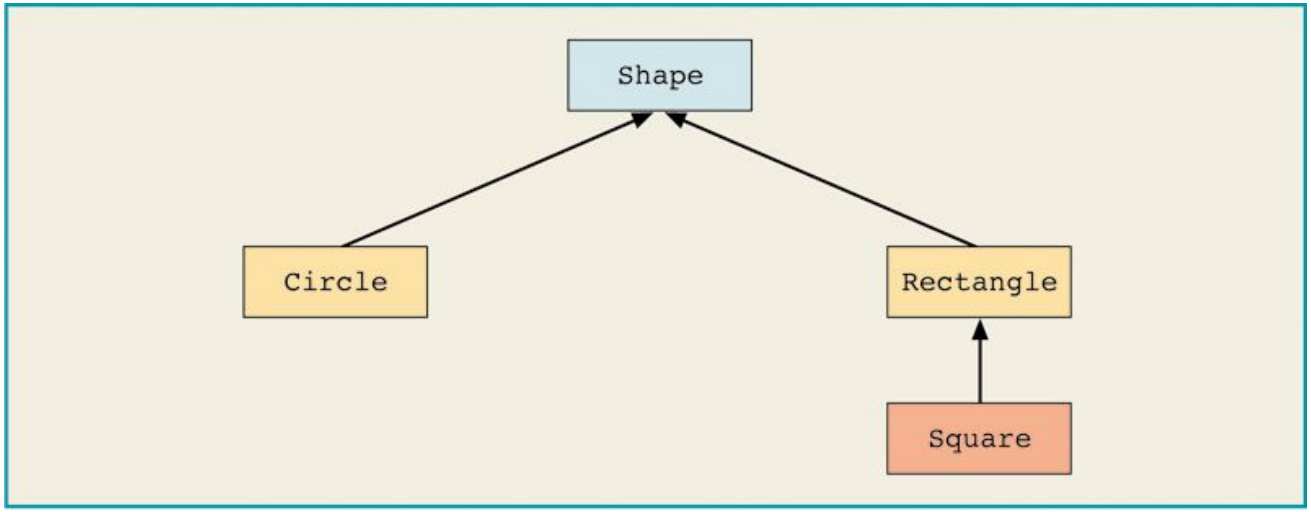



**Figure 11-1** Inheritance hierarchy

```
modifier(s) class ClassName extends ExistingClassName {}
```

# *sample classes*

**Superclass**

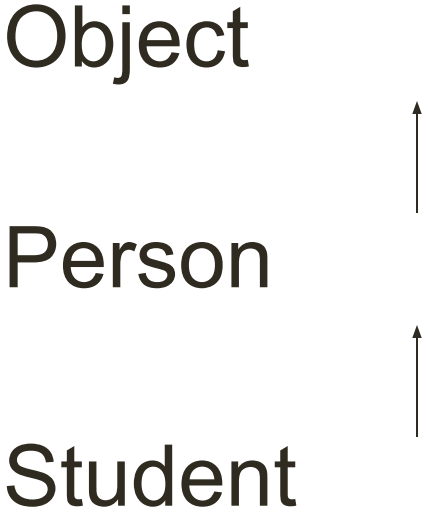
```
public class Person{
    private String name;

    public Person ( ) {
      name = “no_name_yet”;
    }

    public Person ( String initialName ) {
      this.name = initialName;
    }

    public String getName ( ) {
      return name;
    }

    public void setName ( String newName
    ) {
      name = newName;
    }
```

- **Subclass**

```
public class Student extends Person {
    private int studentNumber;

    public Student ( ) {
        super( );
        studentNumber = 0;
    }

    public Student (String initialName, int
        initialStudentNumber) {
        super(initialName);
        studentNumber = initialStudentNumber;
    }

    public int getStudentNumber ( ) {
        return studentNumber;
    }

    public void setStudentNumber (int newStudentNumber )
        {
        studentNumber = newStudentNumber;
}
```

# Inheritance

- Class hierarchy of previous example



# Fields/Methods in Extended Classes

- An object of an extended class contains two sets of variables and methods
  1. fields/methods which are defined locally in the extended class
  2. fields/methods which are inherited from the superclass

What are the fields for a `Student` object in the previous example ?

# Constructors in extended classes

- A constructor of the extended class can invoke one of the superclass's constructors by using the *super* method.
- If no superclass constructor is invoked explicitly, then the superclass's no-arg constructor

  ```
  super( )
  ```

  is invoked automatically as the first statement of the extended class's constructor.
- Constructors are not methods and are NOT inherited.

# Three phases of an object's construction

- When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a three-phase construction:
  - invoke a superclass's constructor
  - initialize the fields by using their initializers and initialization blocks
  - execute the body of the constructor
- The invoked superclass's constructor is executed using the same three-phase constructor. This process is executed recursively until the `Object` class is reached

# To Illustrate the Construction Order. .

```
class X {
     protected int xOri = 1;
     protected int whichOri;
     public X() {
          whichOri = xOri;
     }
}
```

```
class Y extends X {
     protected int yOri = 2;
     public Y() {
          whichOri = yOri;
     }
}
```

```
Y objectY = new Y();
```

| Step | what happens | xOri | yOri | whichOri |
|---|---|---|---|---|
| 0 | fields set to default values | 0 | 0 | 0 |
| 1 | Y constructor invoked | 0 | 0 | 0 |
| 2 | X constructor invoked | 0 | 0 | 0 |
| 3 | Object constructor invoked | 0 | 0 | 0 |
| 4 | X field initialization | 1 | 0 | 0 |
| 5 | X constructor executed | 1 | 0 | 1 |
| 6 | Y field initialization | 1 | 2 | 1 |
| 7 | Y constructor executed | 1 | 2 | 2 |

# Overloading and Overriding Methods

- *Overloading*: providing more than one method with the same name but different parameter list
  - overloading an inherited method means simply adding new method with the same name and different signature

- O*verriding*: replacing the superclass's implementation of a method with your own design.
  - both the parameter lists and the return types must be exactly the same
  - if an overriding method is invoked on an object of the subclass, then it's the subclass's version of this method that gets implemented
  - an overriding method can have different access specifier from its superclass's version, but only wider accessibility is allowed
  - the overriding method's `throws` clause can have fewer types listed than the method in the superclass, or more specific types

# Accessibility and Overriding

- a method can be overridden only if it's accessible in the subclass
  - `private` methods in the superclass
    - cannot be overridden
    - if a subclass contains a method which has the same signature as one in its superclass, these methods are totally unrelated
  - `package` methods in the superclass
    - can be overridden if the subclass is in the same package as the superclass
  - `protected`, `public` methods
    - always will be

Not as that simple as it seems!

## Sample classes

```
package P1;

public class Base {
   private void pri( ) { System.out.println("Base.pri()"); }
           void pac( ) { System.out.println("Base.pac()"); }
   protected void pro( ) { System.out.println("Base.pro()"); }
   public void pub( ) { System.out.println("Base.pub()"); }

   public final void show( ) {
       pri();  pac();  pro();  pub(); }
}
```

```
package P2;

import P1.Base;

public class Concrete1 extends Base {
   public void pri( ) { System.out.println("Concrete1.pri()"); }
   public void pac( ) { System.out.println("Concrete1.pac()"); }
   public void pro( ) { System.out.println("Concrete1.pro()"); }
   public void pub( ) { System.out.println("Concrete1.pub()"); }
}
```

```
Concrete1 c1 = new Concrete1();
c1.show( );
```

**Output?**

```
Base.pri()
Base.pac()
Concrete1.pro()
Concrete1.pub()
```

# *Sample classes (cont.)*

```
package P1;

import P2.Concrete1;

public class Concrete2 extends Concrete1 {
   public void pri( ) { System.out.println("Concrete2.pri()"); }
   public void pac( ) { System.out.println("Concrete2.pac()"); }
   public void pro( ) { System.out.println("Concrete2.pro()"); }
   public void pub( ) { System.out.println("Concrete2.pub()"); }
}
```

---

```
Concrete2 c2 = new Concrete2();
c2.show( );
```

**Output?**

```
Base.pri()
Concrete2.pac()
Concrete2.pro()
Concrete2.pub()
```

# Hiding fields

- Fields cannot be overridden, they can only be hidden
- If a field is declared in the subclass and it has the same name as one in the superclass, then the field belongs to the superclass cannot be accessed directly by its name any more

# Polymorphism

- Java allows us to treat an object of a subclass as an object of its superclass. In other words, a reference variable of a superclass type can point to an object of its subclass.
  - when you invoke a method through an object reference, the *actual class of the object* decides which implementation is used
  - when you access a field, the declared type of the reference decides which implementation is used

# Polymorphism

- Late binding or dynamic binding (run-time binding):
  - Method to be executed is determined at execution time, not compile time.
- The term polymorphism means to assign multiple meanings to the same method name.
- In Java, polymorphism is implemented using late binding.
- These reference variables have many forms, that is, they are polymorphic reference variables. They can refer to objects of their own class or to objects of the classes inherited from their class.

## *Example Classes*

```
class SuperShow {
   public String str = "SuperStr";

   public void show( ) {
    System.out.println("Super.show:" + str);
   }
}

class ExtendShow extends SuperShow {
   public String str = "ExtendedStr";

   public void show( ) {
    System.out.println("Extend.show:" + str);
   }

   public static void main (String[] args) {
      ExtendShow ext = new ExtendShow( );
      SuperShow sup = ext;
      sup.show( ); //1
      ext.show( ); //2
      System.out.println("sup.str = " + sup.str);//3
      System.out.println("ext.str = " + ext.str);//4
   }
}
```

methods invoked through object reference

field access

Output?

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

# Type compatibility

- Java is a *strongly typed* language.
- Compatibility
  - when you assign the value of an expression to a variable, the type of the expression must be compatible with the declared type of the variable: it must be the same type as, or a subtype of, the declared type
  - `null` object reference is compatible with all reference types.

# Type conversion (1)

- The types higher up the type hierarchy are said to be *wider*, or *less specific* than the types lower down the hierarchy. Similarly, lower types are said to be *narrower*, or *more specific*.
- *Widening conversion*: assign a subtype to a supertype
  - can be checked at compile time. No action needed
- *Narrowing conversion:* convert a reference of a supertype into a reference of a subtype
  - must be explicitly converted by using the *cast* operator

# Type conversion (2)

- Explicit type casting: a type name within parentheses, before an expression
  - for widening conversion: not necessary and it's a safe cast

    e.g.
    ```
    String str = "test";
    Object obj1 = (Object)str;
    Object obj2 = str;
    ```

  - for narrowing cast: must be provided and it's an unsafe cast

    e.g.
    ```
    String str1 = "test";
    Object obj = str1;
    String str2 = (String)obj;
    Double num = (Double)obj;
    ```

    - If the compiler can tell that a narrowing cast is incorrect, then a compile time error will occur
    - If the compiler cannot tell, then the run time system will check it. If the cast is incorrect, then a ClassCastException will be thrown

# Type conversion (3)

- Type testing: you can test an object's actual class by using the `instanceof` operactor

e.g.
```
if ( obj instanceof String)
 {
        String str2 = (String)obj;
    }
```

# `protected` members

- To allow subclass methods to access a superclass field, define it `protected`. But be cautious!
- Making methods `protected` makes more sense, if the subclasses can be trusted to use the method correctly, but other classes cannot.

# What `protected` really means

- Precisely, a `protected` member is accessible within the class itself, within code in the same package, and it can also be accessed from a class through object references that are of at least the same type as the class – that is , references of the class's type or one of its subtypes

```
public class Employee {
  protected Date hireDay;
  . . .
}

public class Manager extends Employee {
  . . .
  public void printHireDay (Manager p) {
 System.out.println("mHireDay: " + (p.hireDay).toString());
  }
  // ok! The class is Manager, and the object reference type is also Manager
  public void printHireDay (Employee p) {
    System.out.println("eHireDay: " + (p.hireDay).toString());
  }
  // wrong! The class is Manager, but the object reference type is Employee
  // which is a supertype of Manager
  . . .
}
```

# Protected Example

```
package A;

public class Employee {
  protected Date hireDay;
  . . .
}


package B;

public class Manager extends Employee {
  . . .
  public void printHireDay (Manager p) {
 System.out.println("mHireDay: " + (p.hireDay).toString());
  }
  // ok! The class is Manager, and the object reference type is also Manager
  public void printHireDay (Employee p) {
    System.out.println("eHireDay: " + (p.hireDay).toString());
  }
  // wrong! The class is Manager, but the object reference type is Employee
  // which is a supertype of Manager
  . . .
}
```

# Object: the ultimate superclass

- The `object` class is the ultimate ancestor: every class in Java extends `Object` without mention
- Utility methods of `Object` class
  - `equals`: returns whether two object references have the same value
  - `hashCode`: return a hash code for the object, which is derived from the object's memory address. Equal objects should return identical hash codes
  - `clone`: returns a clone of the object
  - `getClass`: return the run expression of the object's class, which is a `Class` object
  - `finalize`: finalize the object during garbage collection
  - `toString`: return a string representation of the object

# The `class` `Object`: Equivalent Definition of a Class

```
public class Clock
{
    //Declare instance variables as given in Chapter 8
    //Definition of instance methods as given in Chapter 8
    //...
}
```

```
public class Clock extends Object
{
  //Declare instance variables as given in Chapter 8
  //Definition of instance methods as given in Chapter 8
  //...
}
```

# `final` Methods and Classes

- Declaring variables **`final`**
  - Indicates they cannot be modified after declaration
  - Must be initialized when declared
- Declaring methods **`final`**
  - Cannot be overridden in a subclass
  - **`static`** and **`private`** methods are implicitly **`final`**
  - Program can inline **`final`** methods
    - Actually inserts method code at method call locations
    - Improves program performance
- Declaring classes **`final`**
  - Cannot be a superclass (cannot inherit from it)
  - All methods in class are implicitly **`final`**

# This and super keywords